Statement of Problem

In order to more closely estimate the number of moves that are required from a given game board state of a 16-piece puzzle to the solution, we construct a neural network. While there are other, less complicated methods of calculating an estimate answer of how many moves are left, such as the Manhattan distance method, which we will be using later to weight against the accuracy of our neural network's predictions, creating a more accurate estimation system would prove to be much more useful. The significance of a more accurate estimator could lead to the development of more accurate path estimators in general, leading to more feasible solutions to otherwise infeasible problems such as the traveling salesman problem.

Restrictions and Limitations

As stated in the project documentation itself, creating a neural network for estimating the number of moves left in a 16 piece puzzle is not a good use of a neural network. On top of this basic limitation of neural networks in general, it should be noted that evaluating the neural network's estimations is only possible for states in which we already know the minimum possible moves to solve. In this sense, rather than trying to bridge between the P and NP problem spaces (as finding the minimum possible moves for an n-piece puzzle is in the NP space), we are merely creating an estimator that mimic such a bridge for this specific type of puzzle, which can only work for up to 28 moves in our case.

Approach

I approached this problem using Python version 3. The neural network was constructed using the Google TensorFlow backend libraries and Keras. Numpy was also used for some additional computational functions.

Layers

The neural network was constructed with 1 input layer, 1 output layer, and 2 hidden layers. Here is a short outline of the layer structure:

1. 240 input neurons, using hyperbolic tangent.
2. 120 neurons, hyperbolic tangent.
3. 60 neurons, hyperbolic tangent.
4. 29 output neurons, sigmoid for our output.

Experiments

When experimenting with the network, I kept the layers the same for all tests. I wanted to make sure whatever I was doing wasn't dependent on the overall structure of the network itself. I used Stochastic Gradient Descent as my optimizer and Mean Squared Error for my loss function.

In training the neural network, I fed a batch of 1000 states for each file (in which I either fed 1000 states or all the states in the current file if there were less. I then experimented with varying epochs / batch sizes.

Sample Run

Here are some sample runs of the program running both through Jupyter's HTTP server interface and through VPN on the compute server on a plain Python script.

```
        pos_data = pos_data[1:]

        state_pos = []

        for p in pos_data:
            state_pos.append(p[1])

        testing_target_pos = reduce(generate_pos, pos_data, [])

        testing_target.append(format_man_dist(man_dist_state(state_pos, testing_target_pos)))

        counter += 1
        data = f.read(8)

#print(testing_target)
```

Evaluating our test data

In [30]:
```
# Evaluate accuracy

loss_and_metrics = model.evaluate(np.array(testing),np.array(testing_target), batch_size=1000)

# Generating predictions:

predictions = model.predict(np.array(testing), batch_size=1000)
```

```
20000/20000 [==============================] - 0s 2us/step
```

In [31]:
```
output = []

for p in range(len(predictions)):
    if np.argmax(testing_target[p]) < 18:
        output.append(100*((18 - (28 - np.argmax(predictions[p]))) / (18 - np.argmax(testing_target[p]))))
    else:
        output.append(0)

#for i in range(len(output)):
#    print(output[i])

print(np.array(output).mean())

print(loss_and_metrics)

print(model.metrics_names)
```

```
13.4831845238
[0.18453341573476792, 0.0095500001683831211]
['loss', 'acc']
```

Python on compute.cse.tamu.edu

```
1000/1000 [==============================] - 0s 13us/step - loss: 0.2233 - acc: 0.0000e+00
Epoch 5/5
1000/1000 [==============================] - 0s 13us/step - loss: 0.2229 - acc: 0.0000e+00
25
Epoch 1/5
1000/1000 [==============================] - 0s 12us/step - loss: 0.2166 - acc: 0.0850
Epoch 2/5
1000/1000 [==============================] - 0s 12us/step - loss: 0.2162 - acc: 0.0850
Epoch 3/5
1000/1000 [==============================] - 0s 12us/step - loss: 0.2158 - acc: 0.0850
Epoch 4/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2154 - acc: 0.0850
Epoch 5/5
1000/1000 [==============================] - 0s 13us/step - loss: 0.2150 - acc: 0.0870
26
Epoch 1/5
1000/1000 [==============================] - 0s 12us/step - loss: 0.2186 - acc: 0.0020
Epoch 2/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2182 - acc: 0.0020
Epoch 3/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2178 - acc: 0.0020
Epoch 4/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2174 - acc: 0.0020
Epoch 5/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2170 - acc: 0.0020
27
Epoch 1/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2216 - acc: 0.0010
Epoch 2/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2213 - acc: 0.0010
Epoch 3/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2209 - acc: 0.0010
Epoch 4/5
1000/1000 [==============================] - 0s 10us/step - loss: 0.2206 - acc: 0.0010
Epoch 5/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2202 - acc: 0.0010
28
Epoch 1/5
1000/1000 [==============================] - 0s 12us/step - loss: 0.2148 - acc: 0.0640
Epoch 2/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2143 - acc: 0.0670
Epoch 3/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2139 - acc: 0.0670
Epoch 4/5
1000/1000 [==============================] - 0s 11us/step - loss: 0.2135 - acc: 0.0680
Epoch 5/5
1000/1000 [==============================] - 0s 13us/step - loss: 0.2130 - acc: 0.0690
446342/446342 [==============================] - 2s 5us/step
87.4963046167
[0.21731308226626603, 0.038374161657919237]
['loss', 'acc']

[shadow8t4]@compute ~/CSCE420/nn420-private> (19:53:32 12/07/17)
::
```

Results & Analysis

First, I ran a test on the default input of my network. I used 1000 states or all states for each file, 5 epochs, and 1000 in a batch. I received varying results each time I ran the program, with very little consistency. On testing predictions, I would receive higher “improvement values” (we define percentage improvement in the project documentation to be percentage closer to actual number of moves versus the manhattan distance estimate) the farther away I got from the higher number estimates.

I continued to run tests, changing epochs and batch sizes. In one of my final tests, I changed epochs to 8 and batch size to 2000, here is a full list of the data results from that evaluation:

11
1938/1938 [==============================] - 0s 2us/step
Percentage possible improvement: 131.991744066
loss 0.167936483834
acc 0.0175438595376

12
5808/5808 [==============================] - 0s 2us/step
Percentage possible improvement: 153.83953168
loss 0.169155473757
acc 0.0118801652392

13
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 91.605
loss 0.167931690812
acc 0.0303999996744

14
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 115.414166667
loss 0.166890135407
acc 0.0186000001617

15
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 98.1758333333
loss 0.168728539348

acc 0.0131000000983

16
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 34.4588333333
loss 0.168766576052
acc 0.010100000212

17
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -53.8332380952
loss 0.17060739994
acc 0.0102000000887

18
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -0.692416666667
loss 0.169354042411
acc 0.00440000006929

19
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -29.2496706349
loss 0.171234123409
acc 0.00790000013076

20
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 23.3521944444
loss 0.170418299735
acc 0.007200000179

21
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -31.1126269841
loss 0.171738886833
acc 0.0229000000283

22
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 9.1381010101
loss 0.173329897225

acc 0.00890000014333

23
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -43.6177330447
loss 0.174675036967
acc 0.0130999999936

24
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -3.36664033189
loss 0.176320441067
acc 0.0120000001742

25
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: -1.81612357087
loss 0.177039775252
acc 0.00450000000419

26
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 33.477459596
loss 0.174290961027
acc 0.017100000754

27
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 4.31057944833
loss 0.173010015488
acc 0.00779999999795

28
10000/10000 [==============================] - 0s 2us/step
Percentage possible improvement: 30.1904823787
loss 0.174648806453
acc 0.024500000081

Note that I only test 10,000 states, this is to cut down on computation time.

Conclusions

The current model of the neural network that is presented ranges from unstable to almost unusable in a practical sense. It should be noted that due to the nature of the neurons being weighted differently, it would make sense that the connections we trained more often (eg: those with more states, such as the higher number moves left states) would be more influential during predictions. However, it’s worth noting that even when fixing this issue, possibly by manually weighting the tests per file on something like a 1/1000 scalar, it would not fix the overall issue of the inconsistency of the network overall.

Future Research

As stated in the conclusion, it would improve the network to have some 1/n type weighting scheme during training. Additionally, it was brought up to me by a colleague that a possible reworking of the structure of the layers could prove to be helpful, as the input layer could instead take 15 16-bit inputs telling the current tile rather than a 240 neuron layer spread out for each possible occurrence of any 0 to 15 tile piece.

Instructions on running (README)

If you are running the program through the compute.cse.tamu.edu servers, you will need to make sure you run using the command:

```
python3 neural_network.py
```

In order to install the necessary libraries to run the program, you will also need to run these commands:

```
wget "https://bootstrap.pypa.io/get-pip.py"
python3 get-pip.py --user
python3 -m pip install --user tensorflow
python3 -m pip install --user keras
python3 -m pip install --user numpy
```

If you are using a virtual desktop or are logged in on campus somewhere, you should be able to also view the Jupyter Notebook that is included, which has more detailed comments. You will just need to install an additional package.

python3 -m pip install --user jupyter
python3 -m notebook nn_puzzle_solver.ipynb

In case you are unable to access the notebook, I've included an HTTP version of the notebook as well.

The Program

```
from functools import reduce
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from os.path import join

# Used to format our input binary state.

def format_input(acc, elem):
      hex_elem = (elem - (elem >> 4 << 4))
      for x in range(16):
      if x == hex_elem:
            acc.append(1)
      else:
            acc.append(0)
      hex_elem = (elem >> 4) % 16
      for x in range(16):
      if x == hex_elem:
            acc.append(1)
      else:
            acc.append(0)
      return acc

# Calculate Manhattan distance between two points.

def man_dist(x, y):
      for a, b in zip(x, y):
      a_one, a_two = x
      b_one, b_two = y

      return (abs(a_one - b_one) + abs(a_two - b_two))
```

```
# Calculate Manhattan distance between each set of two points in a list.

def man_dist_state(x, y):
      return sum(man_dist(a, b) for a, b in zip(x, y))

# Used to format the positions we parsed from our binary input.

def format_pos(acc, elem):
      hex_elem = (elem[1] - (elem[1] >> 4 << 4))
      if hex_elem == 0:
      acc.append((hex_elem, (3,3)))
      else:
      acc.append((hex_elem, ((15 - ((elem[0]) * 2)) % 4,int((15 - 
((elem[0]) * 2)) / 4))))
      hex_elem = (elem[1] >> 4) % 16
      if hex_elem == 0:
      acc.append((hex_elem, (3,3)))
      else:
      acc.append((hex_elem, ((15 - ((elem[0]) * 2 + 1)) % 4,int((15 - 
((elem[0]) * 2 + 1)) / 4))))

      return acc

# The title of this function is slightly misleading.
# I'm simply generating a list of positions that each
# puzzle piece in the current parsed state SHOULD be at.
# I organize this in order of the pieces as they were
# parsed so the two lists line up perfectly.

def generate_pos(acc, elem):
      if(elem[0] == 0):
      acc.append((3,3))
      else:
      acc.append((((elem[0] - 1) % 4), (int((elem[0] - 1)/4))))

      return acc

# Used to format our ending Manhattan distance into a format
# that can be compared with our 29 output neurons.

def format_man_dist(elem):
```

```
	acc = []
	for x in range(28, -1, -1):
	if x == elem:
		acc.append(1)
	else:
		acc.append(0)
	return acc


target = []

for i in range(29):
	filename = join('/pub/faculty_share/daugher/datafiles/data/' + str(i)
+ 'states.bin')

	# Debugging to print the current file from which states are being
parsed.
	#print(i)
	temp = []

	with open(filename, 'rb') as f:
	data = f.read(8)
	counter = 0

	while(data and counter < 2000):
		temp.append(format_man_dist(i))

		data = f.read(8)
		counter += 1

	target.append(temp)

#print(target[28][500])

# Sets up a Sequential model, Sequential is all
# that should need to be used for this project,
# considering that it will only be dealing with
# a linear stack of layers of neurons.

model = Sequential()

# Adding layers to the model.
```

```
model.add(Dense(units=240, activation='tanh', input_dim=240))
model.add(Dense(units=120, activation='tanh'))
model.add(Dense(units=60, activation='tanh'))
model.add(Dense(units=29, activation='sigmoid'))

# Configure the learning process.

model.compile(optimizer='sgd',
      loss='mean_squared_error',
      metrics=['accuracy'])


for i in range(29):
      filename = join('/pub/faculty_share/daugher/datafiles/data/' + str(i)
+ 'states.bin')

      # Debugging to print the current file from which states are being
parsed.
      print(i)

      with open(filename, 'rb') as f:
      data = f.read(8)
      counter = 0
      training = []

      while(data and counter < 2000):
            bin_data = reduce(format_input, list(data), [])
            bin_data.reverse()
            bin_data = bin_data[16:]

            training.append(bin_data)

            data = f.read(8)
            counter += 1

            #print(training[0])
      # Train the network.

      model.fit(np.array(training), np.array(target[i]), epochs=8,
batch_size=2000)
      #model.train_on_batch(np.array(temp), np.array(target))
```

```
# Used for testing data

for i in range(11, 29):
      filename = join('/pub/faculty_share/daugher/datafiles/data/', str(i)
+ 'states.bin')

      print(i)

      with open(filename, 'rb') as f:

      for i in range(2000):
            data = f.read(8)

      data = f.read(8)

      counter = 0

      testing = []

      testing_target = []

      while(data and counter < 10000):
            bin_data = reduce(format_input, list(data), [])
            bin_data.reverse()
            bin_data = bin_data[16:]

            testing.append(bin_data)

            pos_data = reduce(format_pos, enumerate(list(data)), [])
            pos_data.reverse()
            pos_data = pos_data[1:]

            state_pos = []

            for p in pos_data:
                  state_pos.append(p[1])

            testing_target_pos = reduce(generate_pos, pos_data, [])

            testing_target.append(format_man_dist(man_dist_state(state_pos,
testing_target_pos)))
```

```
            counter += 1
            data = f.read(8)


      # Evaluate accuracy

      loss_and_metrics =
model.evaluate(np.array(testing),np.array(testing_target), batch_size=1000)

      # Generating predictions:

      predictions = model.predict(np.array(testing), batch_size=1000)

      output = []

      for p in range(len(predictions)):
            if np.argmax(testing_target[p]) < 18:
                  output.append(100*((18 - (28 -
np.argmax(predictions[p]))) / (18 - np.argmax(testing_target[p]))))
            else:
                  output.append(0)

      #for i in range(len(output)):
      #     print(output[i])

      print("Percentage possible improvement: ", np.array(output).mean())

      print(model.metrics_names[0], loss_and_metrics[0])

      print(model.metrics_names[1], loss_and_metrics[1])
```

Bibliography

"Keras: The Python Deep Learning Library." *Keras Documentation*, keras.io/.